Brief Announcement: Extending Transactional Memory with Atomic Deferral

Tingzhe Zhou
Lehigh University
tiz214@lehigh.edu

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Michael Spear
Lehigh University
spear@lehigh.edu

ABSTRACT

Atomic deferral is a language-level mechanism for transactional memory (TM) that enables programmers to move output and long-running operations out of a transaction's body without sacrificing serializability: the deferred operation appears to execute as part of its parent transaction, even though it does not make use of TM.

We introduce the first implementation of atomic deferral, based on transaction-friendly locks; describe enhancements to its API; and demonstrate its effectiveness. Our experiments show that atomic deferral is useful for its original purpose of moving output operations out of transactions, and also for moving expensive library calls out of transactions. The result is a significant improvement in performance for the PARSEC dedup kernel, for both software and hardware TM systems.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**; **Concurrent computing methodologies**;

KEYWORDS

Transactional Memory, Concurrency, Synchronization, I/O

ACM Reference format:
Tingzhe Zhou, Victor Luchangco, and Michael Spear. 2017. Brief Announcement: Extending Transactional Memory with Atomic Deferral. In *Proceedings of SPAA '17, Washington DC, USA, July 24-26, 2017,* 3 pages.
https://doi.org/10.1145/3087556.3087600

1 INTRODUCTION

Transactional memory (TM) [6] simplifies the task of writing correct, scalable code. With TM, a programmer designates code regions as "transactions". A run-time system, which employs custom hardware [8] (HTM) and/or compiler-generated software instrumentation [4, 7] (STM), monitors the low-level memory accesses of transactions, and allows concurrent transactions to execute simultaneously as long as their memory accesses do not conflict. If transactions conflict, the run-time system rolls back some subset of transactions and retries them. However, some operations (e.g., I/O) cannot be rolled back, and others are expensive, so that rolling them back wastes a lot of work (and increases the likelihood of conflict by extending the transaction). Programmers should avoid including such operations in transactions.

One approach for allowing long-running and I/O operations to be included in a transaction is to designate such a transaction as irrevocable [9, 10, 13], so that it cannot be aborted. When an irrevocable transaction runs, no other transaction may run. This ensures that the irrevocable transaction will not encounter a conflict, and thus will not abort. Consequently, it is safe for it to perform I/O, and the capacity constraints of HTM, and overheads of STM, need not apply.

As an alternative, a limited set of operations can be delayed until after a transaction commits [2, 9]. These "deferred" operations typically cannot access shared memory, and other threads might observe that the operations run *after* the transactions that requested them. By incorporating special "sentinel" locks into the operating system, Volos et al. [11] enabled broad support for transactional system calls, to include input and output. To perform output from a transaction, the operating system would lock the corresponding file descriptor, and make a copy of the data to write. When the transaction committed, the copied data would be written, and then the lock released. In this manner, the output would appear atomic with the transaction.

In a previous workshop paper, we introduced atomic deferral [14]. Atomic deferral is a language-supported mechanism for deferring output and other long-running operations. Like Volos's work, atomic deferral makes use of locks. However, the locks are a userspace construct, fully visible to the programmer. Programmers can associate one of these locks with any object, and then the compiler ensures that transactions who access those objects "subscribe" to the corresponding lock. To defer an operation, a transaction acquires the locks associated with the objects used by the operation. When the transaction commits, the locks will be held, and thus mutual exclusion is guaranteed for the deferred operation. After the deferred operation completes, the locks are released.

As a purely userspace construct, atomic deferral does not perform system calls during a transaction. Thus HTM transactions can use atomic deferral, whereas they cannot use sentinel locks. Furthermore, atomic deferral is not limited to system calls: it can be used by programmers to defer arbitrary operations, and it provides a path for programmers to avoid copying shared data before using it in deferred operations.

In this brief announcement, we describe our experience implementing atomic deferral and using it in the PARSEC dedup [1] kernel. The output operations in dedup are a known cause of serialization and poor scaling [12]. With atomic deferral, we moved these output operations out of transactions, resulting in a program with no mandatory irrevocability. However, the compression and decompression operations in dedup then become a bottleneck: they

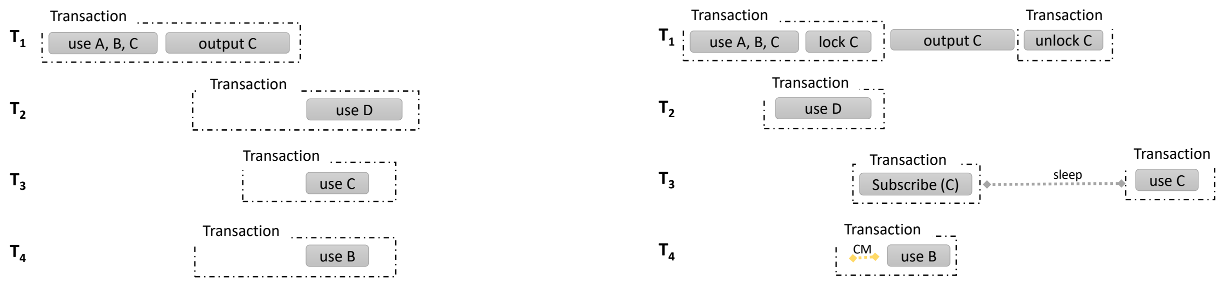

Figure 1: Motivation for atomic deferral. On the left, T_1's transaction outputs object C, which prevents other transactions, even unrelated ones, from making progress. On the right, C is locked, and then the output of C is deferred until after the transaction commits. The use of locking and deferral of the output of C enables the operations by threads T_2 and T_4 to progress more quickly, without violating serializability.

caused HTM transactions to exceed cache capacity, and STM transactions to incur instrumentation delays. Since atomic deferral is not limited to system calls, we also deferred these operations. After doing so, transactional dedup scaled well, performing on par with carefully tuned locks.

2 ATOMIC DEFERRAL

Atomic deferral makes use of special transaction-friendly spin locks (`TxLocks`), which, in turn, make use of `retry`-based condition synchronization [5]. `TxLocks` can be acquired from within transactions, and when an attempt to acquire a `TxLock` fails, the enclosing transaction aborts, rolls back, and does not try again until the `TxLock` is released. A transaction may access the data associated with a `TxLock` as long as the lock is not held. To do so, it *subscribes* to the lock. That is, before accessing the data, the transaction reads the lock to ensure that it is not held. If the lock is subsequently acquired, the subscribing transaction will automatically abort. If the lock is held by another thread when a transaction attempts to subscribe, the transaction will abort, and will not attempt to run again until the `TxLock` is released.

Armed with `TxLocks`, atomic defer is able to transform the behavior on the left side of Figure 1 to the behavior on the right side. In the figure, time flows from left to right. On the left side, the output of C causes the transaction by T_1 to become irrevocable. Due to irrevocability, the transaction by T_2 cannot start, even though it does not conflict with T_1's transaction. The transactions by threads T_3 and T_4 also must wait for the output to complete. However, these transactions have overlapping accesses with T_1's transaction, and might not be able to execute concurrently with T_1's transaction.

On the right side, T_1's transaction defers the `output` operation on object C. Before the transaction commits, it acquires C's `TxLock`. After committing, the output is performed, and then C is unlocked. Since the output is not performed within a transaction, T_1's transaction does not become irrevocable. Thus T_2's non-conflicting transaction does not delay. Similarly, T_4's transaction need only delay as long as is necessary for the contention manager to ensure that T_1 and T_4's transactions can both make progress. If both only read B, then T_4's transaction would not need to wait. T_3's transaction subscribes to the `TxLock` associated with C. When T_1's transaction acquires the lock, it causes T_3's transaction to abort, and T_3 to sleep until the lock is released. At that point, T_3 can wake and access C using a transaction.

Note that the execution remained serializable with atomic defer. On the right side of the figure, the result is equivalent to one in which T_1 completes, then T_2, and then T_3 and T_4 in either order. In particular, T_4 appears to complete after T_1, because it sees any updates to B; because it does not access C, it can run concurrently with the operations on C, yet order after T_1. If it had accessed C, then like the operation in T_3, it would have slept until T_1's transaction committed. In our extended API to atomic deferral, `TxLocks` are reentrant, and a request to defer an operation names all of the objects whose `TxLocks` must be held during the execution of the operation. In this manner, a thread may defer multiple operations, and yet still achieve serializability through two-phase locking [3]: All locks are acquired before a thread's transaction commits, and then no further locks are acquired; no locks are released until after the transaction commits; and after each deferred operation, only the corresponding locks are released, with reentrancy ensuring that they are not released too early.

In addition, we observe that there is no reason that the deferred operation on C must be an output operation. As long as the remainder of T_1's initial transaction does not access C, an arbitrary computation on C can be deferred without sacrificing atomicity.

3 PERFORMANCE EVALUATION

In their study of transactional condition synchronization [12], Wang et al. found that PARSEC's dedup kernel [1] ceased to scale when transactions replaced locks. Dedup is a pipeline application, where the final stage performs output while holding a lock; Wang's version replaces that lock with an irrevocable transaction. When the irrevocable transaction executes, it must serialize all concurrent transactions. We rewrote dedup's output operation to use atomic deferral. With the change, irrevocability ceased to cause performance degradation, but the benchmark still did not scale well.

The performance of dedup with this change appears in Figure 2, as the "+DeferIO" curves. The results were measured on a system with a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. This CPU supports Intel's TSX extensions for HTM, includes 8 GB of RAM, and runs a Linux 4.3 kernel. Atomic defer was added to GCC 5.3.1. Results are the average of 5 trials.

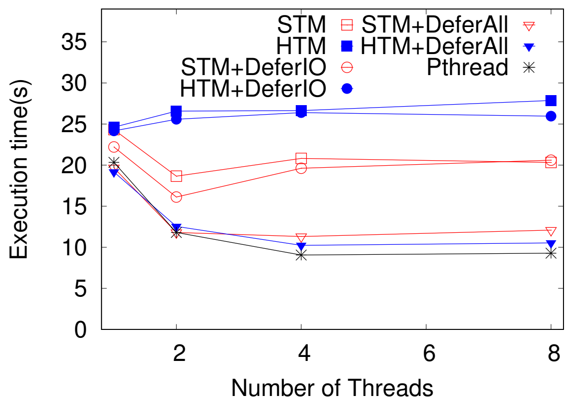

Figure 2: Using atomic deferral in PARSEC dedup

During profiling, we discovered that the `Compress` function was marked as `pure`, because it does not access any shared memory. Marking the function `pure` indicates to the compiler that the function can be run without instrumentation, lacks side effects, and can be run from a non-irrevocable context even when the compiler cannot prove that irrevocability is not needed. `Compress` is a long-running function, and in HTM, it accesses more memory than can be tracked by the HTM; the HTM execution serializes whenever a call to `Compress` exceeds the capacity of the hardware. In STM, a transaction executing `Compress` causes other transactions to delay in their commit operation. While the run-time behaviors are different, the consequence is the same: when one transaction calls `Compress`, other transactions cannot make progress.

`Compress` is amenable to atomic deferral, and the impact is profound. In HTM, the transaction ceases to overflow hardware capacity, and serialization is avoided. In STM, compression ceases to impede concurrent commits, and concurrent threads can make forward progress. In Figure 2, we see that the "+DeferAll" curves for both HTM and STM now compete with pthread locks, representing a 1.7x speedup for STM and 2.7x speedup for HTM. On a 36-thread multi-chip TSX machine (not shown), both HTM and STM performed equivalently to the baseline Pthread locks at all thread counts.

4 CONCLUSIONS AND FUTURE WORK

In this brief announcement, we presented our experience extending atomic deferral, implementing it, and using it on the PARSEC dedup kernel. In our experience, atomic deferral did not significantly complicate the program: the total lines of code changed in dedup were fewer than a dozen, and reasoning about what could and could not be deferred was straightforward. Using atomic deferral allowed transactions to perform output without serializing, and to move other long-running operations out of transactions. The result was a dramatic improvement in the performance of the transactional PARSEC dedup benchmark.

As future work, we are interested in tools for automatically transforming output operations into deferred operations, and studying the relationship between atomic deferral and nested transactions. We are also interested in crafting a more formal correctness argument, which may influence the use of transaction-friendly locks in a greater range of workloads.

Acknowledgments

We thank Dave Dice and Tim Harris for their advice and guidance during the conduct of this research. At Lehigh University, this work was supported in part by the National Science Foundation under Grant CAREER-1253362, and through financial support from Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Toronto, ON, Canada.

[2] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. 2006. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*.

[3] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* 19, 11 (1976), 624–633.

[4] Free Software Foundation. 2012. Transactional Memory in GCC. (2012). http://gcc.gnu.org/wiki/TransactionalMemory.

[5] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*. Chicago, IL.

[6] Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. San Diego, CA.

[7] ISO/IEC JTC 1/SC 22/WG 21. 2015. Technical Specification for C++ Extensions for Transactional Memory. (May 2015). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf

[8] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. Portland, OR.

[9] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*. Nashville, TN, USA.

[10] Michael Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. 2008. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*. Portland, OR.

[11] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael Swift, and Adam Welc. 2009. xCalls: Safe I/O in Memory Transactions. In *Proceedings of the EuroSys2009 Conference*. Nuremberg, Germany.

[12] Chao Wang, Yujie Liu, and Michael Spear. 2014. Transaction-Friendly Condition Variables. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. Prague, Czech Republic.

[13] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. Munich, Germany.

[14] Tingzhe Zhou and Michael Spear. 2016. The Mimir Approach to Transactional Output. In *Proceedings of the 11th ACM SIGPLAN Workshop on Transactional Computing*. Barcelona, Spain.